Performance of the MP_Lite message-passing library on Linux clusters

Dave Turner, Weiyi Chen and Ricky Kendall
Scalable Computing Laboratory, Ames Laboratory, USA

Abstract

MP_Lite is a light-weight message-passing library designed to deliver the maximum performance to applications in a portable and user-friendly manner. It supports a subset of the most commonly used MPI commands, or it can be run using its own simpler syntax. The high level of performance is achieved by implementing a core set of communication primitives in a very clean fashion. MP_Lite resorts to extra buffering only when necessary. Its small size also makes it an ideal tool for performing message-passing research. On Unix clusters, MP_Lite can run as a thin layer on top of TCP, providing all the performance that TCP offers for applications that can take advantage of this restricted mode. The default signal based interrupt driven mode is more robust and still provides around 90% of the TCP performance on Gigabit Ethernet networks. Current research involves the development of an MP_Lite module for the M-VIA OS-bypass library. Bypassing the OS reduces the message latency and streamlines the flow of data to the network hardware. Reducing the memory-to-memory copies puts much less strain on the memory bus, allowing more of the full bandwidth of the PCI bus to be realized when coupled with Gigabit-level networking. MP_Lite also has the capability of striping data across multiple network interfaces per computer. If done at the TCP level, dual Fast Ethernet interfaces double the performance for PC clusters at a minimal increase to the overall cost. Current work with the M-VIA module allows effective channel bonding of at least three Fast Ethernet interfaces per machine for PC clusters, providing a scalable networking solution for low cost clusters.

1 Introduction

The MPI standard [1-2] and its many implementations [3-4] provide a consistent set of functions available on almost every type of multiprocessor system. This standardization of the syntax and functionality for message-passing systems provides a uniform interface from the application to the underlying communication network. Programs can now use a common set of communication calls that perform the same in a very wide variety of multiprocessor environments.

While the MPI standard provides portability, the native communication libraries and underlying communication layers that the MPI implementations are built upon typically offer much better performance. Since the performance differences may be as great as a factor of two in both bandwidth and latency, there is a strong temptation to use the native libraries at the expense of the portability that MPI offers. For PC/workstation clusters, the inter-processor communication rate is already much slower than in traditional massively parallel processing (MPP) systems so it is vital to squeeze every bit of performance out of the message-passing system.

The inefficiencies in MPI implementations can come from several sources. The MPI standard itself imposes many challenges to implementers. The choices to support a wildcard (MPI_ANY_SOURCE) for the source in a receive function, out-of-order messages, and byte mismatches between send and receive pairs can all lead to extra buffering and complicate internal queuing systems.

MPICH and LAM are the two most common public domain MPI implementations. Both provide support for a few dozen multiprocessor environments, which necessarily requires a multi-layered programming approach. This can easily lead to extra buffering of all messages. The additional memory-to-memory copies reduce the communication bandwidth and increase the latency.

The ultimate goal of the research presented here is to understand exactly where the inefficiencies come from and to make MPI and its implementations more efficient. The source codes for MPICH and LAM are freely available, but they are very complex codes that have a steep learning curve for users who wish to make modifications. They are geared for production environments, not message-passing research.

The MP_Lite library [5] is smaller and much easier to work with. It is ideal for performing message-passing research that can eventually be used to improve full MPI implementations and possibly influence the MPI standard itself. The core subset of MPI functions supported is enough to handle many MPI applications, and represents enough of the MPI standard to provide a reasonable test for message-passing research. In other words, the benefits from research with MP_Lite should also be realizable within any full MPI implementation.

2 The MP_Lite message-passing library

MP_Lite is a light-weight message-passing library that delivers the performance of the underlying communication layer while maintaining portability, all in a user-friendly manner. It was born out of the need to squeeze every bit of performance from the inter-processor communication between nodes in PC/workstation clusters, where the Fast Ethernet and even Gigabit Ethernet speeds are much slower than communication rates in traditional MPP systems. It has also found a home on MPP systems like the Cray T3E where there is a large difference between the performance of MPI and the native SHMEM communication library.

MP_Lite provides the core set of functionality that MPI and most other message-passing libraries [6-7] have, but does so in a streamlined manner that allows it to deliver the maximum performance to the application layer. The small size of the code and its clean nature make it easy to modify, making it an ideal research tool for studying inter-processor and inter-process communications.

The library is designed to be user-friendly at many levels. The small size makes it easy to take and install anywhere, compiling in well under a minute. The library can be recompiled into several debugging levels to assist in tracking down communication related bugs. A simple profiling system makes it easy to monitor and tune the performance of codes. When the communication functions block for longer than a user definable panic time, the message-passing system times out and dumps the current state of the message queues to provide help in deciphering any problems.

MP_Lite has its own simpler syntax that can be used instead of the MPI syntax. There is no support for communicators to subdivide the nodes for global operations, and the data types are handled by using slight variations of the function calls for each data type. This syntax provides much more readable code, and getting rid of the abstraction also has the benefit of eliminating the need for the *INCLUDE "mpif.h"* statement in each FORTRAN subroutine that has MPI function calls. Below is an example of a global summation of an integer array of length 10, and a send function of 20 doubles to node 5 using a message tag of 0.

```
MP_iSum( iarray, 10 );

MP_dSend( darray, 20, 5, 0 );
```

Most users will probably choose to use the MPI syntax when using MP_Lite. There is absolutely no loss in performance when using MPI wrappers in MP_Lite. The trade-off with MP_Lite is that it only provides a basic set of communication primitives, which is a subset of the many offerings of MPI. This is enough to support a large portion of the scalable codes around, and it is easy to add more functionality as needed. However, MP_Lite would not be appropriate for more complex codes that make use of the abstraction of communicators to deal with multiple levels of parallelism. MP_Lite will always be kept as a light-weight, high performance alternative to complete MPI implementations.

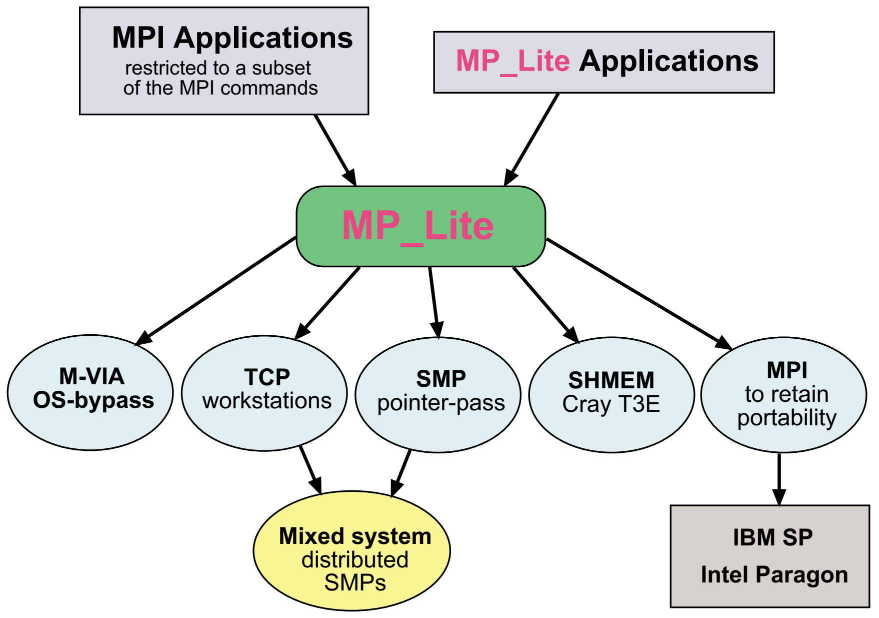

Figure 1: Diagram of the structure of MP_Lite showing the modules that have been implemented.

2.1 How MP_Lite works

Figure 1 shows schematically how the MP_Lite library is organized. Applications can use the MP_Lite function calls directly, or they can access MP_Lite using MPI without modification as long as the code is restricted to the subset of MPI functions supported.

The intermediate level contains support for all the collective operations, including global sum, min, and max operations as well as a barrier synchronization. There are also Cartesian coordinate functions, I/O functions, advanced timing functions, and support for a variety of other ongoing areas of research.

These functions are all built upon the point to point communications implemented in each of the architecture specific modules. These basic communication primitives are written on top of the underlying communication layer. For Linux clusters, this includes four possibilities. Communications within an SMP node can occur using TCP or using shared-memory segments.

There are then two modes that can be used for TCP communications; a synchronous mode that is just a thin layer on top of TCP, and an asynchronous or interrupt driven mode that is more robust. Current work is in progress on an M-VIA [8] module that bypasses the operating system to provide lower latency and better bandwidth. There is also a module written on the Cray SHMEM library allows MP_Lite to provide twice the performance of MPI on the Cray T3E.

Each of these modules implements the blocking and asynchronous send and receive commands, as well as the initialization and cleanup for the message-passing system. The implementations vary in detail, but the resulting functionality provided to the higher-level communication functions is the same. Send and receive queues must be carefully managed to minimize the buffering of messages when possible. The best performance is delivered when the application programmer cooperates by preposting receives, which allows the messages to be put directly into user space without any additional buffering. Out-of-order messages from the same node, posting receives where the number of bytes is larger than the actual message sent, and using the MPI_ANY_SOURCE wildcard for the source in a receive function can all hurt the performance by imposing additional buffering.

Applications are launched using a simple *mprun* shell script that is rsh/ssh-based for Unix systems, or defaults to mpprun or mpirun when appropriate. The *mprun* script caches all launch data to a configuration file, allowing subsequent launches to utilize the same information without needing to retype it each time.

Each node generates a *.nodeX* log file for each run where *X* is the relative node number. These log files contain initialization data, and any detailed warning or error messages that may occur during a run. If a run is interrupted manually with a control-C, the current state of the message queues will be dumped to the log files. If a message blocks past the user definable panic time, the state of the queues will also be dumped and the run will be gracefully aborted in most cases.

2.2 The synchronous TCP mode

On Unix clusters, the MP_Lite module is written using the sockets interface to the TCP/IP layer. The simplest method of implementing a message-passing system is to increase the size of the TCP send and receive buffers and use them to buffer all messages. This provides the maximum throughput by minimizing additional buffering of messages. This is a very natural approach, since all incoming and outgoing packets must be buffered at this layer anyway, and it avoids the additional memory-to-memory copying and buffer management at a higher level that can hurt the performance. It is simply a very thin layer above TCP, and therefore provides all the performance that TCP offers even for Gigabit-level networks.

In this mode, the asynchronous send and receive functions do nothing more than log the message information, letting the wait function handle the actual data transfer. This is therefore a fully synchronous system. A system of message queues is used to manage any message buffering needed due to out-of-order messages imposed either by the use of message tags or from the use of a wildcard for the source node in a receive. Receiving a message from an unknown source requires all TCP buffers to be actively searched which can greatly affect the overall performance since any message sitting in a TCP buffer must be pushed to a receive buffer so that subsequent messages can be checked for a match. Achieving high performance therefore requires the application programmer to avoid practices that cause extra message buffering when in time-critical areas of the code. The same is true for all MPI implementations.

While providing the maximum throughput and being easy to implement, this synchronous approach has one obvious deficiency. Since it is a fully synchronous system, it can lock up if nodes send more information than can fit into the TCP buffers. This approach is therefore only useful when the user can guarantee that the message traffic will not exceed the size of the TCP buffers. Setting the TCP buffers to around 1 MB each can make this system usable for many applications, but could tax memory for large clusters. In any case, the responsibility is put upon the end user to insure that the application can run in the given environment.

2.3 The asynchronous interrupt-driven mode

The default mode for running MP_Lite on Unix is a more robust method that still passes along the majority of the performance that TCP offers. In this method, the asynchronous send and receive functions begin the data transfers then return control to the application without waiting for completion. For an asynchronous send, a SIGIO interrupt signal is generated when data is transferred out of the TCP send buffer. A signal handling routine traps all SIGIO interrupts and services the given message by pushing more data into the appropriate TCP send buffers. An asynchronous receive will read all available data in the TCP receive buffer, then return control to the application. A SIGIO interrupt is generated when more data arrives, and the signal handling routine services the active receive queues by reading the new data.

The wait function simply blocks until the signal handling routine completes the data transfer. The blocking send and receive functions are then just the corresponding asynchronous versions followed by a blocking wait function. The same message queuing system described in the previous section tracks all message buffering. One difference from the synchronous approach is that the source node does not block on a send. The wait function instead uses a *malloc()* to allocate a send buffer then copies the data from the application memory space. The signal handling routine will then complete the send without the source node blocking, avoiding all possibility of a lockup condition that can occur in the synchronous approach.

This asynchronous module is therefore a robust system since it cannot suffer from the lockup condition that could happen with the synchronous approach. The interrupt-driven communications are not always as efficient as the synchronous communications, but performance is very good even when using the default TCP buffer sizes which are typically around 64 kB. Increasing the TCP buffer sizes can improve the performance since less buffering is required and fewer signals need to be serviced.

2.4 The M-VIA module

M-VIA [8] is a modular implementation of the Virtual Interface Architecture that is being developed for Linux systems by the National Energy Research Scientific Computing Center. VIA [9-10] is an industry standard that enables high-performance communications between nodes in a cluster through bypassing the operating system. Bypassing the operating system can streamline the exchange of data between computers by reducing the memory-to-memory copies to a minimum. However, M-VIA must then duplicate the needed functionality that it is replacing in the OS, by providing a reliable system for packetizing and delivering data across unreliable networks such as Ethernet.

The MP_Lite M-VIA module is programmed using the VIA API. While internally much different than the TCP modules, this module provides the same functionality to the upper layers of the MP_Lite library and uses the same type of message queues.

Each node pre-posts a number of buffers to receive unexpected short messages. Messages smaller than 12 kB are transferred immediately to the destination node through these pre-posted buffers. This Eager protocol assumes the receive side has enough pre-posted buffers to hold the incoming messages. Larger messages are sent using remote dynamic memory access write (RDMA write) mechanisms that are extremely efficient. Each RDMA write requires handshaking between the source and destination nodes. The source node sends a *Request To Send* control message that includes the message size and tag. After registered the destination memory region, the receiving node replies with a *Clear To Send* control message containing the destination buffer address and the registered memory handle. The source node then uses an RDMA write to put the message directly into the destination.

M-VIA provides a much lower latency than TCP, and MP_Lite passes this on to the application layer. The RDMA writes also provide a much higher bandwidth. With TCP, the data being transferred is copied multiple times as it is broken into packets and headers are added. Each memory-to-memory copy strains the main memory bus, which is ultimately what gets saturated. The streamlined approach of M-VIA puts less stress on the memory bus, allowing more of the full bandwidth to be realized.

MVICH [11] is a full MPI implementation that runs on M-VIA. It is being developed by the same group at NERSC that is developing M-VIA. Since M-VIA does not have reliable delivery implemented yet, both MVICH and the MP_Lite M-VIA module are not practical message-passing libraries yet. Implementing reliable delivery will likely degrade the performance.

3 Message-passing performance on Linux systems

All performance evaluations were done on two test clusters. The first contains two 450 MHz PCs running Linux connected by multiple Fast Ethernet interfaces and also a pair of Syskonnect Gigabit Ethernet cards. The second test-bed contains two 500 MHz Compaq DS20 workstations running Linux. All tests used the NetPIPE [12-13] benchmark, which performs a simple ping-pong test between two machines for a range of message sizes.

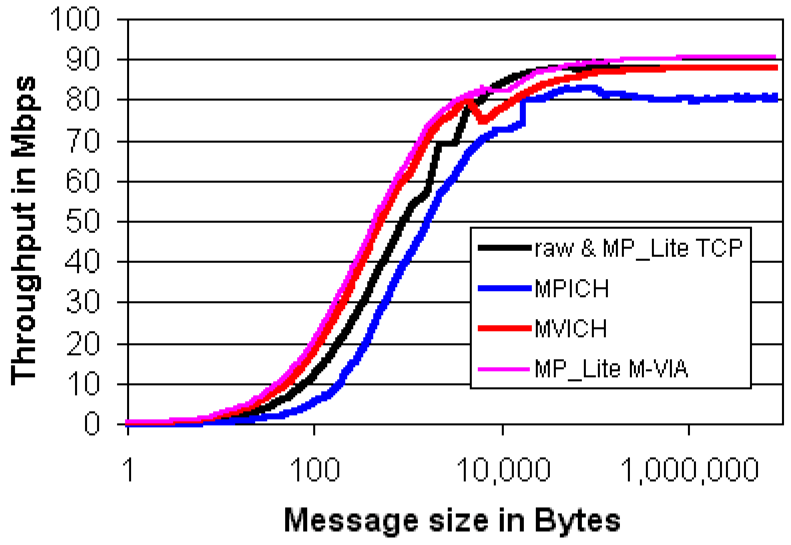

Figure 2: Throughput as a function of message size for raw TCP, the two MP_Lite TCP modules, the MP_Lite M-VIA module, and the MVICH and MPICH libraries.

For slower networks such as Fast Ethernet, any message-passing library can deliver most of the TCP performance adequately. Figure 2 shows that both the synchronous and asynchronous MP_Lite results fall directly on top of the TCP curve, delivering the full 90 Mbps throughput that TCP offers. Full MPI implementations can lose up to 25% of this performance for slower computers, but do not suffer as much for faster PCs and workstations as illustrated by the MPICH curve in this graph.

The major difference for Fast Ethernet networks is that the MP_Lite M-VIA module and MVICH implementation both provide much lower latency at 27 µs and 31 µs respectively compared to MPICH at 121 µs. This can make a large difference for codes that send many small messages, and can also improve the performance for codes that operate in the intermediate range sending message of up to 10 kB in size.

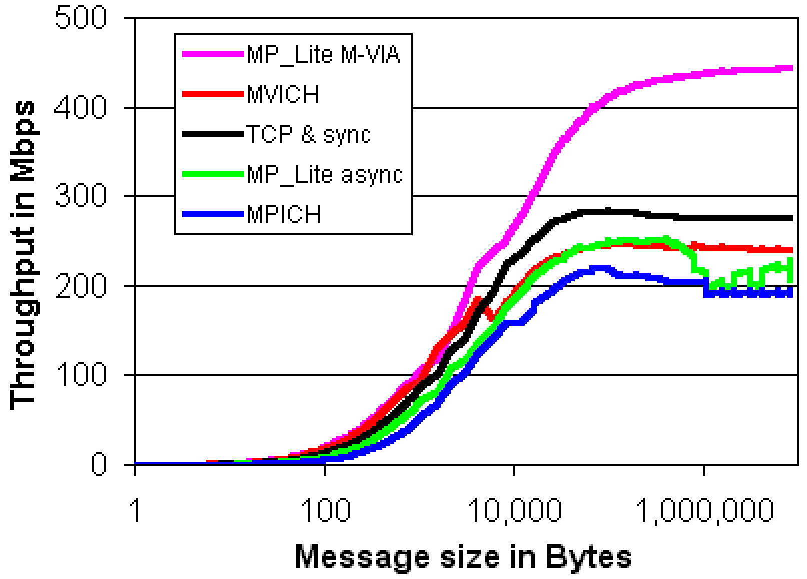

Figure 3: The throughput between Syskonnect Gigabit Ethernet cards on two PCs as a function of message size.

The difference between the message-passing libraries is more evident for faster networks such as Gigabit Ethernet where there is even more strain put on the main memory bus. Figure 3 shows that raw TCP is limited to 280 Mbps, which the synchronous MP_Lite module can deliver to the application layer. MPICH loses about 25% of this raw performance, while the asynchronous MP_Lite module can deliver 90% of the performance for a maximum rate of 250 Mbps.

The MVICH implementation delivers a maximum rate of 245 Mbps, which is better than MPICH. It is still a work in progress and much better results can be expected in the near future. The MP_Lite M-VIA module delivers a maximum bandwidth of 440 Mbps, which is double what MPICH can deliver. The latencies are 34 µs for MP_Lite M-VIA, 39 µs for MVICH, and 127 µs for MPICH.

The Syskonnect Gigabit Ethernet cards support jumbo frames, in which an MTU size of 9000 Bytes is used instead of the standard 1500 Byte MTU. The raw TCP performance for jumbo frames is clearly impressive, topping out at 480 Mbps. However, this requires a switch that supports jumbo frames, which limits its use currently. In the future, it would be nice to run the MP_Lite M-VIA module in conjunction with TCP jumbo frames hardware, but this capability is not coded into M-VIA yet.

4 Channel-bonding on Linux clusters

Channel-bonding is the use of more than one network interface card (NIC) per machine in a PC/workstation cluster. MP_Lite allows messages to be striped across these multiple NICs to increase the potential bandwidth between machines.

The MP_Lite TCP modules do this by simply opening up multiple sockets. Since each socket has its own NIC with a unique IP number, no special setup is needed for the cluster switch. The MP_Lite M-VIA module also supports channel-bonding in a similar manner. This work is still under development, but is showing encouraging results.

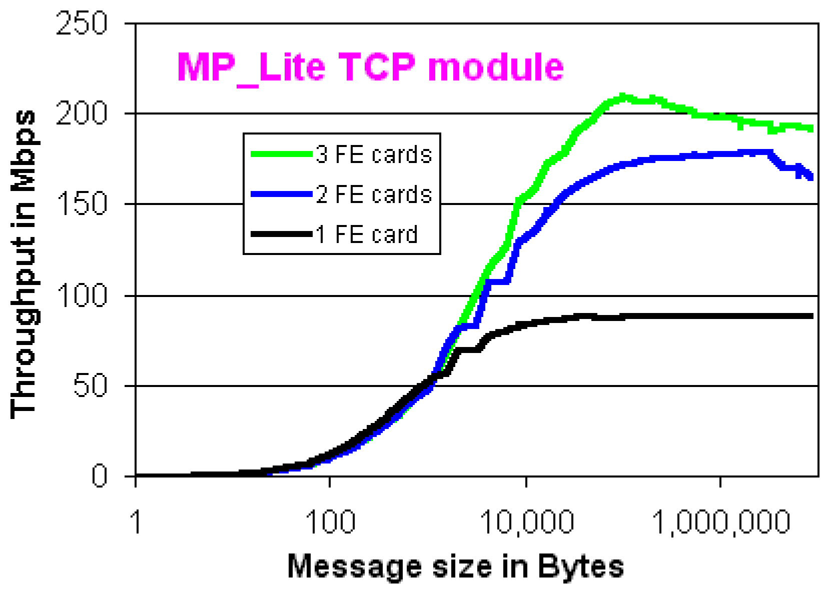

Figure 4: Channel-bonding of 1-3 Fast Ethernet cards between two PCs for the MP_Lite TCP modules.

Channel-bonding two Fast Ethernet interfaces at the TCP level produces an ideal doubling of the bandwidth for PC clusters. Adding a third card clearly provides little benefit. Using Gigabit Ethernet in this environment can provide three times the performance of Fast Ethernet, but doubles the cost of a PC cluster while adding a second Fast Ethernet interface doubles the communication performance for a negligible increase in cost.

The initial MP_Lite M-VIA results in figure 5 are even more impressive. Channel-bonding three Fast Ethernet interfaces provides 260 Mbps, or nearly an ideal tripling of the communication rate. M-VIA still does not have reliable delivery implemented. This is therefore not a usable system yet, and implementing reliable data transfer may affect the performance. However, this performance is very encouraging at this point.

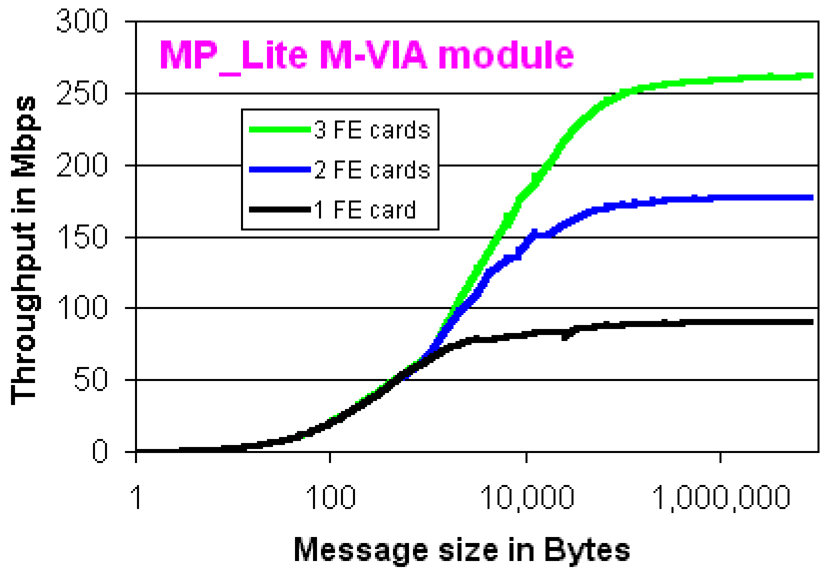

Figure 5: Channel-bonding of 1-3 Fast Ethernet cards between two PCs for the MP_Lite M-VIA module.

5 Conclusions

MP_Lite delivers nearly all the performance that the underlying communication layer offers, and it does so in a portable and user-friendly manner. It provides the most common set of communication functions, and can provide up to twice the performance on many systems for applications that can live within these limitations. Its small size makes it an ideal tool for performing message-passing research.

The TCP modules provide much greater performance than can be attained from existing MPI implementations. This is especially true for faster networks, where MP_Lite can deliver 90-100% of the performance that TCP offers for Gigabit Ethernet networks. The M-VIA work is still in the preliminary stages, but shows great promise of reducing the latency and possibly doubling the effective bandwidth that can be delivered to the application layer for Gigabit-level networks.

Channel-bonding two Fast Ethernet interfaces in a PC cluster can double the communication performance without increasing the cost greatly. When reliable data transfer becomes fully implemented in M-VIA, the use of channel-bonding should be practical for 3-4 Fast Ethernet interfaces per machine, providing scalability to the design of the communication network in low-cost clusters.

Acknowledgements

This project is funded by the DOE MICS department through the Scalable Computing Laboratory at Ames Laboratory. Ames Laboratory is operated by the U.S. Department of Energy by Iowa State University under Contract No. W-7405-Eng-82. The authors would like to than Mike Welcome and Paul Hargrove for several helpful conversations about the M-VIA and MVICH projects.

References

[1] The MPI Standard: http://www.mcs.anl.gov/mpi/
[2] MPI Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8 (3/4) 165-416, 194.
[3] MPICH homepage: http://www.mcs.anl.gov/mpi/mpich/
[4] LAM homepage: http://www.osc.edu/Lam/lam.html
[5] MP_Lite homepage: http://cmp.ameslab.gov/MP_Lite/
[6] PVM homepage: http://www.epm.ornl.gov/pvm/
[7] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam, *PVM: Parallel Virtual Machine*, MIT press, 1994.
[8] M-VIA homepage: http://www.nersc.gov/research/ftg/via/
[9] VI Architecture organization: http://www.viarch.org/
[10] VI Developers Forum: http://www.vidf.org/
[11] MVICH homepage: http://www.nersc.gov/research/FTG/mvich/
[12] NetPIPE homepage: http://www.scl.ameslab.gov/netpipe/
[13] Quinn O. Snell, Armin R. Mikler, and John L. Gustafson. NetPIPE: A Network Protocol Independent Performance Evaluator. *IASTED Conference Paper*.